DØ Run II L2 Programming Rules

R. Moore

Abstract

This document sets basic coding guidelines for writing software for the L2 trigger system. These differ from the general DØ guidelines in that they are optimized for speed over good programming style wherever relevant.

1. Introduction

The level 2 coding environment places severe restrictions on the type of software that can be written. The basic programming language used is C++, however in order to achieve the required performance many features of standard C++ are either not allowed or strictly limited.

2. General Language Limitations

There are several basic limitations placed on the C++ language, many of which are due to the austere programming environment. Violations of these rules in many cases will lead to crashes or bizarre link errors which will be hard to track down – so please follow these rules carefully for your own good!

2.1. No Dynamic Memory Allocation

All memory allocation must be static i.e. no 'new' or 'delete'. The only form or dynamic allocation allowed is from statically allocated arrays managed by the user (and this should be avoided when possible). In the final program the default allocator as well as the C 'malloc' function will be overridden and made to generate errors to prevent accidental dynamic memory creeping in.

The reason for this is two-fold. Firstly the PC164 SDK supplied by Compaq did not support dynamic memory in its earlier versions and even now there is very little control over memory management. Secondly, and more importantly, dynamic memory incurs a significant performance penalty.

2.2. No Run-Time-Type-Identification (RTTI)

This occurs a performance penalty and in almost all cases can be avoided by using static casts instead of dynamic ones, for example use:

```
CFTbuffer *myBuf = (CFTbuffer *)genBuf;
```

rather than:

```
CFTbuffer *myBuf = dynamic_cast<CFTbuffer *>genBuf;
```

The later is safer and a better style but incurs too large an overhead for use in the level 2 even though it would useful in several places!

2.3. No C++-Style Exceptions

The C++ 'throw' and 'catch' commands are not supported in the bare-bones software development kit and hence cannot be used. In the event that we use Linux as the run-time environment their use may be allowed but only if it can be shown that they do not incur a performance penalty. In any case they should only be used for error conditions, not to break out of loops.

2.4. No Standard C++ IO Streams (or other OS services)

Standard C++ IO streams are not available. Instead there is a 'l2ostream' class which will allow formatted IO to the console for debugging purposes only. If this is insufficient then the C 'printf' function must be used. Under no circumstances must the C++ stream header files ever be included in a L2 source file, even if C++ streams are not being used.

Please note that all output to the console must be done only when in debug mode i.e. the macro 'DEBUG' is defined.

2.5. Minimal Use of Virtual Functions

Virtual functions are far more expensive to call than normal functions. For this reason there use should be limited to only those places where they are really needed.

2.6. Inline Functions

Whenever reasonable and possible functions should be inlined. The calling overhead for inlined functions is significantly less than for normal functions. However large functions (more than 3-4 lines) should not be inlined since the compiler may not inline these but instead call it as a normal function giving no performance boost while increasing the size of the executable.

The standard coding guidelines for where to place these functions are the same as for the offline code. They should be placed in the header file after the class definition and not written into the class itself. This makes it far easier to convert them to non-inlined should they grow large enough.

2.7. Variable Types

Standard integer C++ variable types must never be used! Instead the L2 environment supplies types of the form 'int8' or 'uint8' where the number denotes the number of bits and the 'u' denotes an unsigned variable. The allowed numbers of bits are 8, 16, 32 and 64.

There is also another type, 'maddr' which should be used for all memory addresses (when not using a pointer). **Do not assume that all memory addresses or integers are 32 or 64 bits**. Non-integer types 'float', 'double' and 'bool' may be used as per normal C++.

The reasoning behind this is to allow cross-platform compatibility for the simulator (and emulator). When moving from a 64 bit architecture, such as the Alpha, to a 32 bit one, such as Intel, all that is needed is to change the typedefs which are stored in a single include file.

3. Style Guidelines/Rules

Some of these rules may sound rather arbitrary: this is because they are! The purpose behind them is to make us write code with a similar style which will greatly increase the ease of understanding each other's code. Most will be enforced by scripts run on the source code before it is added to the CVS repository. For the moment these tools do not exist so in the meantime please follow the rules as closely as possible: it will greatly reduce your work when the enforcement scripts are written!

3.1. Capitalization of Names

All class names must begin with a capital letter. Variable and function names must begin with a lower case letter. This allows easy identification of classes which, in some circumstances, may be confused as function or variable names.

Mixed case names are strongly encouraged to improved readability. For example: "getMonitorData()" is easier to read than "getmonitordata()" and so is preferred.

3.2. Avoid Abbreviations

Unless a function, variable or class name is excessively long there is no need to abbreviate it. There is no requirement in C++ that names be 6 characters – they can be any length! Abbreviations can be confusing to the user, no matter how obvious they may appear. The two places where you may wish to use abbreviations are in the case of

an excessively long name, for example 20+ characters, or in a frequently used local function variable. Local function variables are not seen by users but should still be named sensibly to aid in reading the code.

For example "readBuffer()" is much clearer than "rdBuf()" and is easier to type than "readBufferFromMemoryToVMEBufferDriver()". This level of detail should be placed in the DOC++ comment and not in the function name!

3.3. Class Layout

Classes should be laid out according to the D0 coding guidelines. This includes placing the public section at the top of the header file, the protected section next and finally the public section. Constructors and initialization functions should be placed at the top of the section they are in.

Inline methods must be declared after the class definition and not inside it. This is to allow easy conversion between inline and non-inline status as well as to avoid cluttering up the class with function code.

3.4. Comments

As a rule there should be *at least* one comment for every line or two of code and a more detailed explanation of complex algorithms should be included at the start of their code block as well as the line-by-line comments. Remember: nobody ever complains that source code has too many comments.

DOC++-style comments should be placed in the class header file to allow automatic documentation of the code. To find out how to write DOC++ comments please read the DOC++ user guide.

Source code with insufficient commentary will not be accepted!

3.5. Source Files

All source files should begin with a comment header stating: the filename, purpose of the file, author and date of creation, the name of the reviewer and a complete revision history with dates and authors. For example the start of the header file for the L2 FIFO template class is given below:

```
//
// File: FIFO.hpp
```

```
// Purpose: L2 FIFO template class
// Created: 09-JAN-1998 by Roger Moore
// Reviewed by:
//
// Comments:
//     Header file for FIFO template class. See
//     Doc++ comments for full documentation.
//
// Revisions:
//  5-JUN-1998 Roger Moore:
//     Added doc++ comments for documentation and
//     updated to  new DEC C++ V6.0 standards.
//
// 26-JUN-1998 Roger Moore:
//     Changed FIFO so that the contents of the
//     internal array are used to determine if the
//     FIFO is full or empty. This simplifies
//    routines and makes the FIFO interrupt safe.
//
```

This exact format should be followed closely as it is planed to use automatic scripts to manage and check the code. It is especially important to document any revisions made to the code.

Each class should have two files associated with it: a header file "<*class name*>.hpp", and a source file "<*class name*>.cpp". Whenever possible "`#include`" pre-processor directives should be placed in the source code file not the header file. This greatly reduces the coupling between classes allowing for a faster compilation.

3.6. Packages

Within level 2 each package's name must begin with '12'. Each package has its own directory with the same name as the package. Inside this directory are the following sub-directories:

'inc': stores all the header ('.hpp') files for the package
'src': stores all the source ('.cpp') files for the package
'obj': where all the object code files are placed during compilation
'depends': all the automatically produced dependency files are placed here

The 'GNUmakefile' lives in the main directory and uses the 'CTEST' standard. Every package *must* have its own self-test suite written before it can be added to the system. At a minimum this test suite must ensure that the basic functionality of the package works and ideally it should test as much of the code as possible, especially in the

case of low-level routines.

Each package should also have a single “<*package name*>.hpp” include file which defines prototypes and constants in the package. By doing this we can cut down the coupling between the packages by having header files include the class prototypes only, not the actual class headers themselves. This limits recompilation, when a class header file is altered, to those classes which directly use the altered class.